ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

(Laboratorio di)
Amministrazione di sistemi

Shell scripting

Marco Prandini

Dipartimento di Informatica – Scienza e Ingegneria

Convenzioni

- Il font `courier` è usato per mostrare ciò che accade sul sistema; i colori rappresentano diversi elementi:

  `rosso per comandi da impartire o nomi di file`

  `blu per l'output dei comandi`

  `verde per l'input (incluse righe nei file di configurazione)`

- Altri colori possono essere usati in modo meno formale per evidenziare parti da distinguere nei comandi o indicazioni importanti nel testo

- I parametri formali sono normalmente scritti in maiuscolo e riportati nello stesso colore nel testo che ne descrive l'utilizzo

Principi di shell scripting

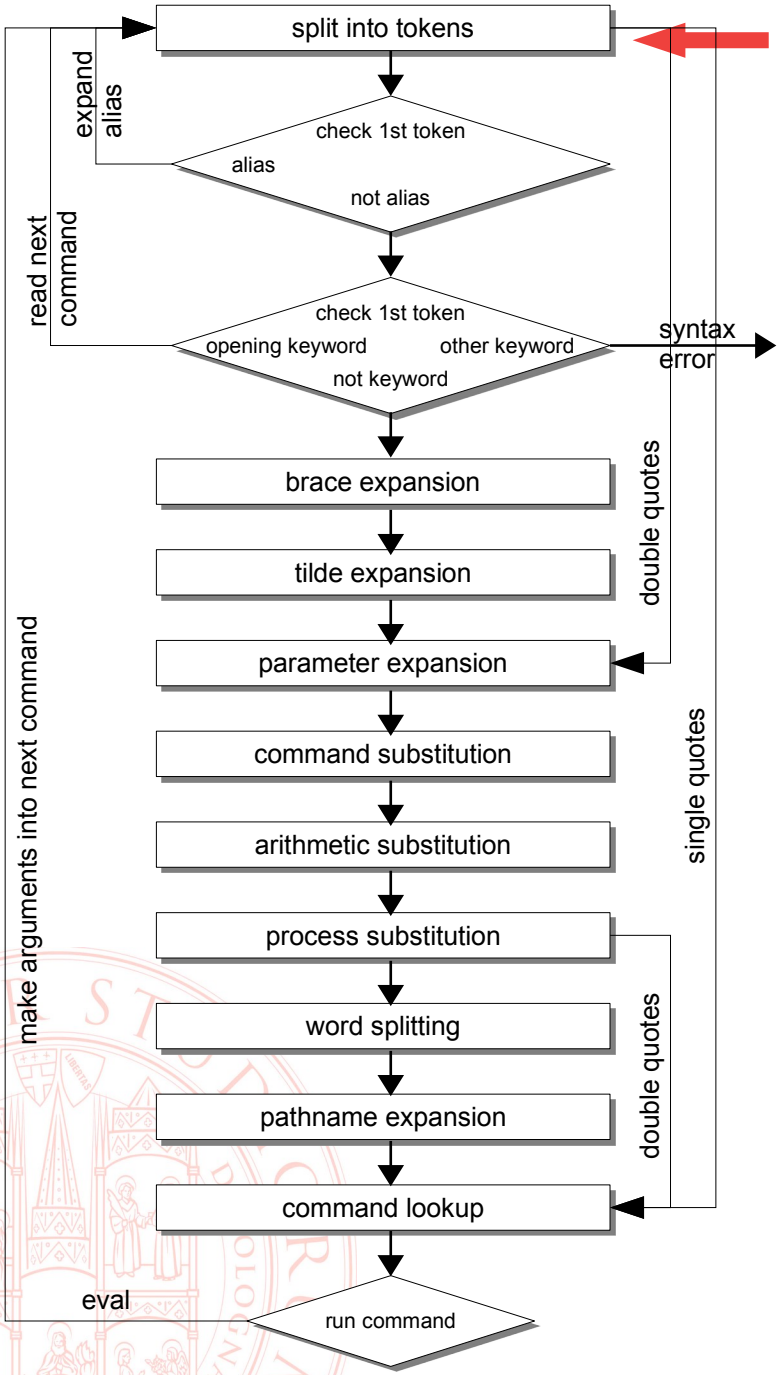
- Bash può essere usata per programmare task da eseguire automaticamente anziché dover impartire comandi a mano
- Ci sono due aspetti importanti da tenere a mente rispetto a un linguaggio di programmazione come C o Java

1) Gli elementi di base gestiti da bash sono file e processi

   bash ha come scopo fondamentale l'avvio di processi, la predisposizione delle comunicazioni tra loro e col filesystem, il controllo dello stato in uscita. È fondamentale pensare sempre, quando si scrive o si analizza una riga di comando, a quali processi verranno eseguiti e a quali file possono essere coinvolti

2) Il linguaggio di bash è interpretato, non compilato

   Il significato dato a molti caratteri è sintattico, non letterale, e la riga di comando effettivamente eseguita risulta da un procedimento, detto espansione, che individua sottostringhe speciali contrassegnate da caratteri speciali, e le sostituisce col risultato di una corrispondente elaborazione



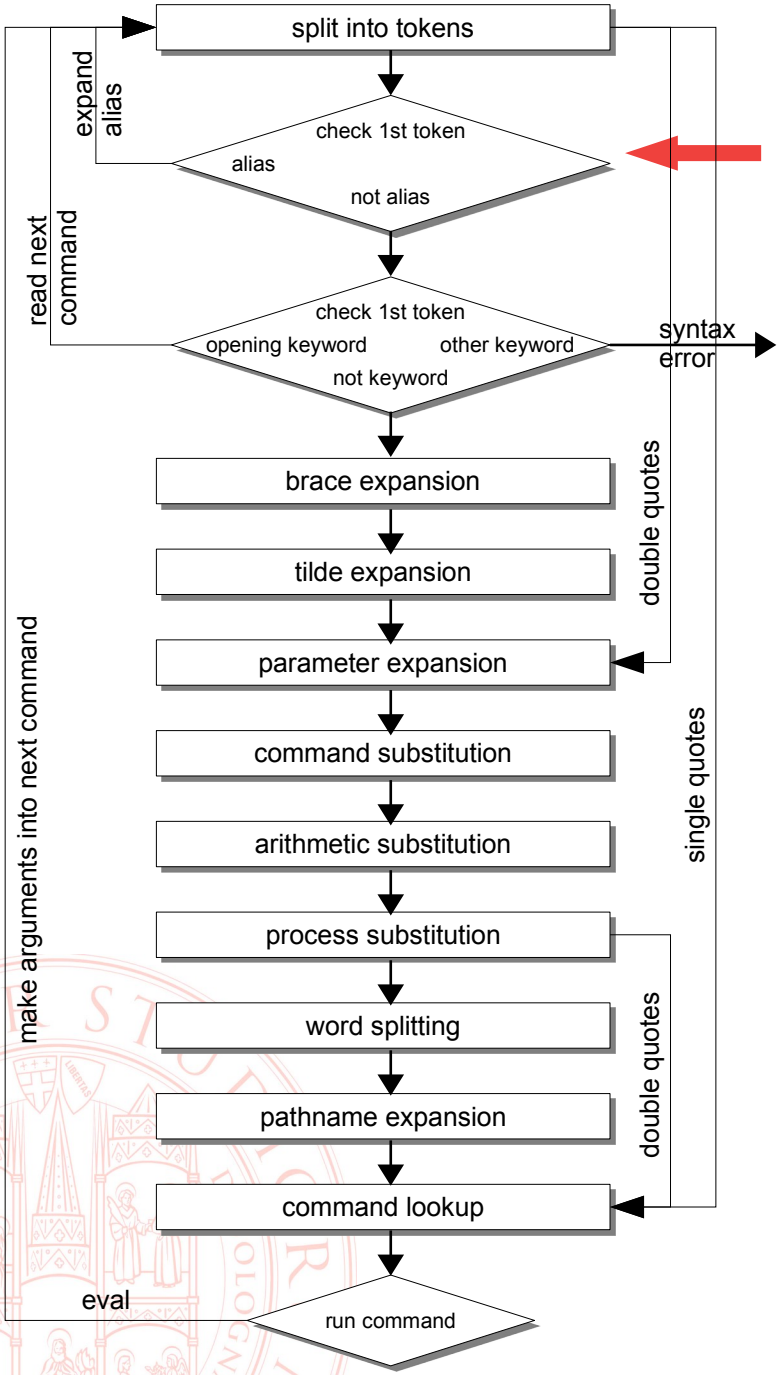
Shell expansion

1. Tokenizzazione

- La riga viene divisa in *token* usando come separatori un elenco fisso di metacaratteri:
  `SPACE TAB NEWLINE`
  `; ( ) < > | &`

- I token possono essere
  - stringhe
  - parole chiave
  - caratteri di ridirezione
  - carattere “ : ”

- Da qui in poi tutti i passi (2-10) sono saltati per le parti di riga racchiuse tra apici singoli



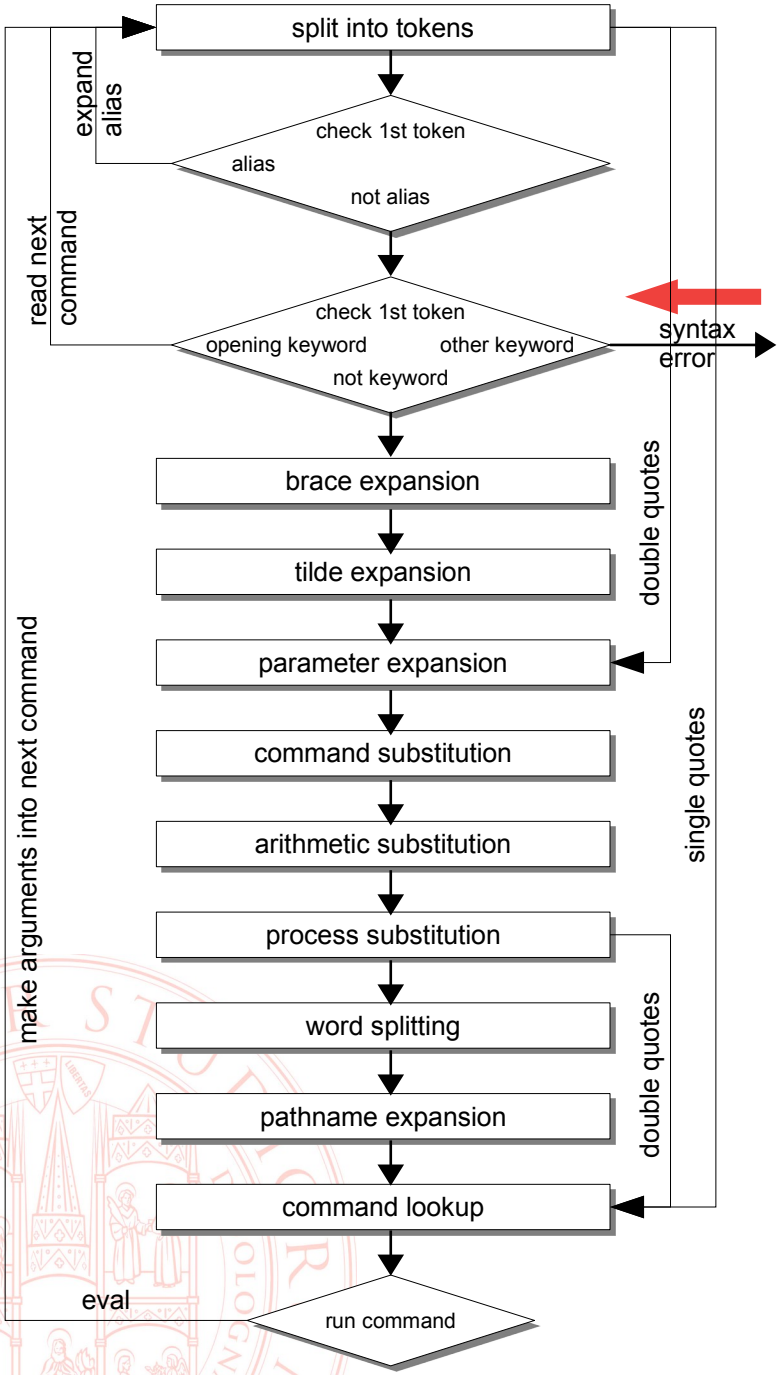
Shell expansion

2. primo token = alias?

- la shell cerca il primo token nella lista degli alias.
- Se lo trova, lo espande e riparte col processing dal punto 1.
  - Si noti che questo consente alias ricorsivi
  - uno stesso alias non verrà mai espanso due volte

    es. `alias ls='ls -l'` non crea loop

- Non eseguito sulle parti di riga racchiuse tra doppi apici



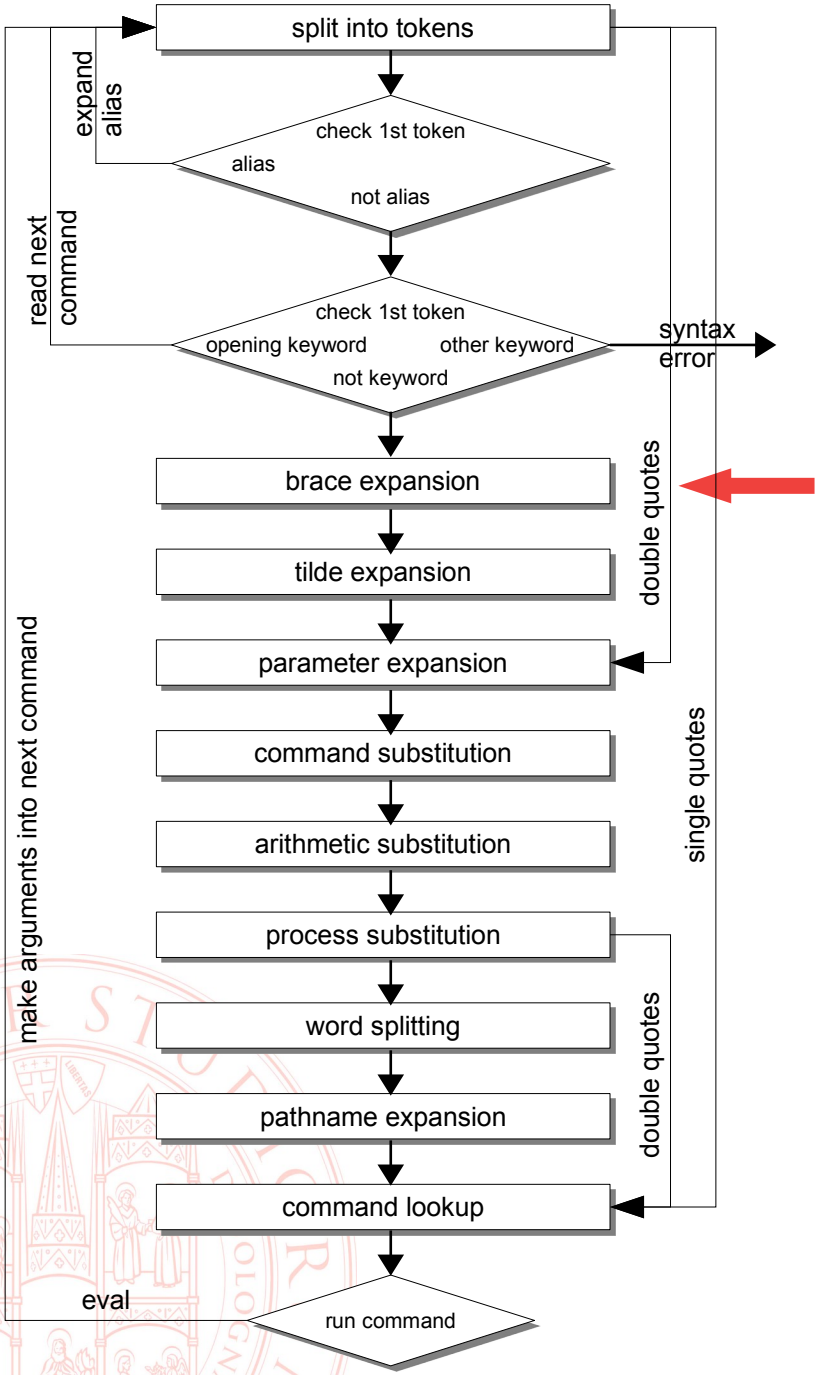
Shell expansion

3. primo token = keyword?

- se il primo token è una parola chiave che dà inizio a un comando composto, ad es.
  - `if`
  - `while`
  - `function`
  - `{`
  - `(`

  la shell predispone l'ambiente per il comando composto e ne va a leggere il primo token

- Non eseguito sulle parti di riga racchiuse tra doppi apici



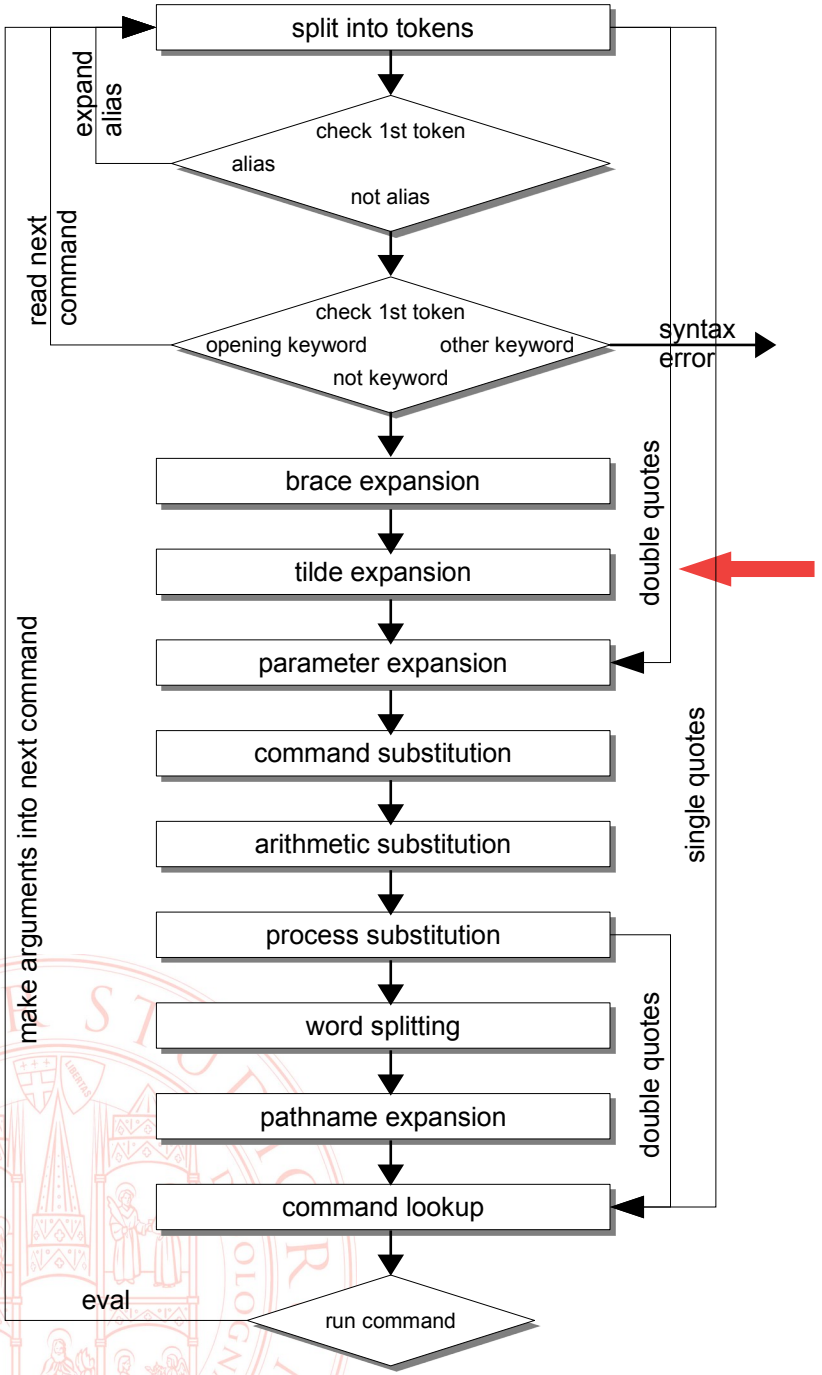
Shell expansion

4. Brace expansion

- Es.

  `Pre{Lista}Post`
  → `PreItem1Post PreItem2Post`
- lista può essere estensiva
  - `{a,pippo,mamma}`
- o sequenza
  - `{min..max[..incr]}`
- … ma esistono moltissime altre brace expansion

- Non eseguito sulle parti di riga racchiuse tra doppi apici

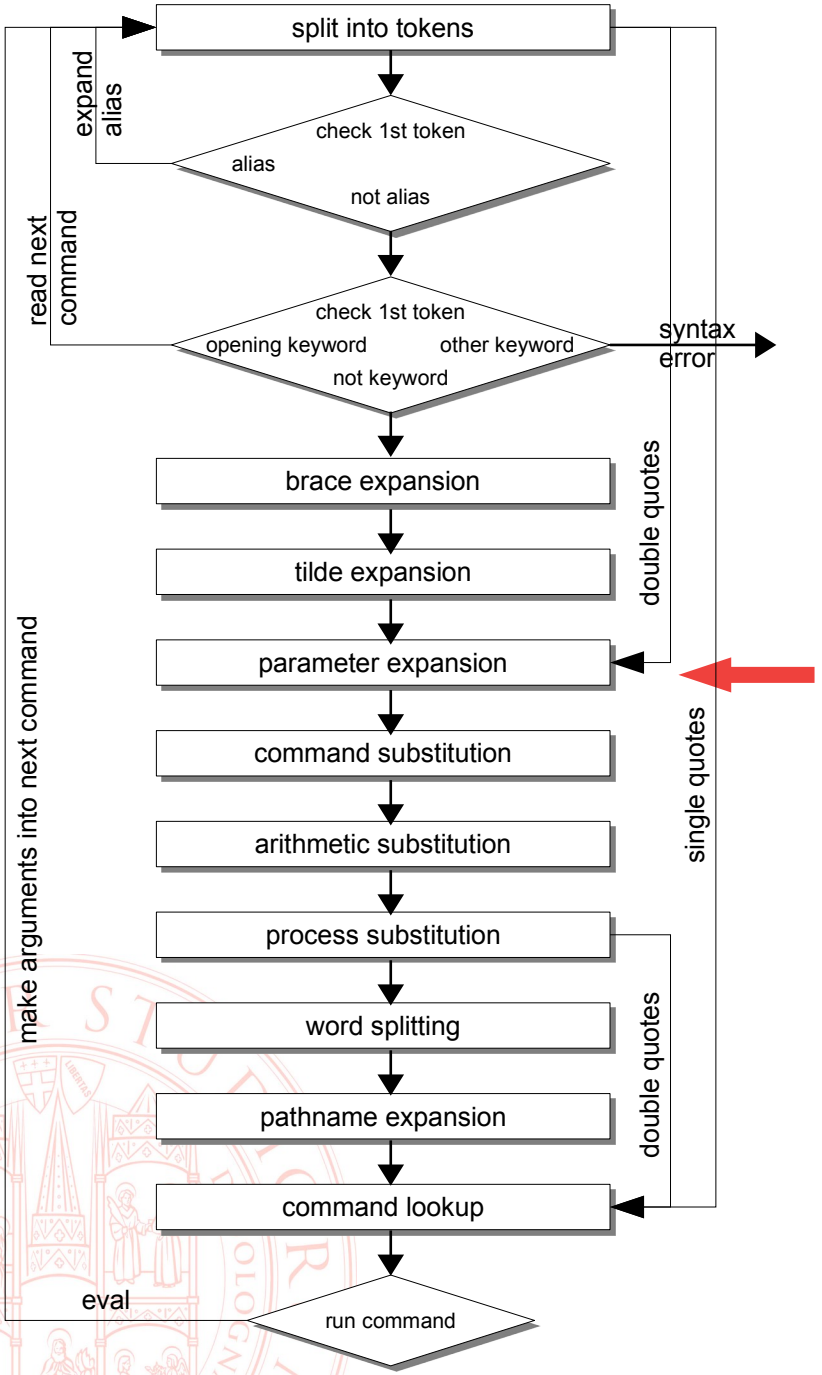


Shell expansion

5. Tilde Expansion

- Se c'è un token nella forma `~username`, viene sostituito con la home directory dell'utente `username` (se username è vuoto, si utilizza l'utente corrente)

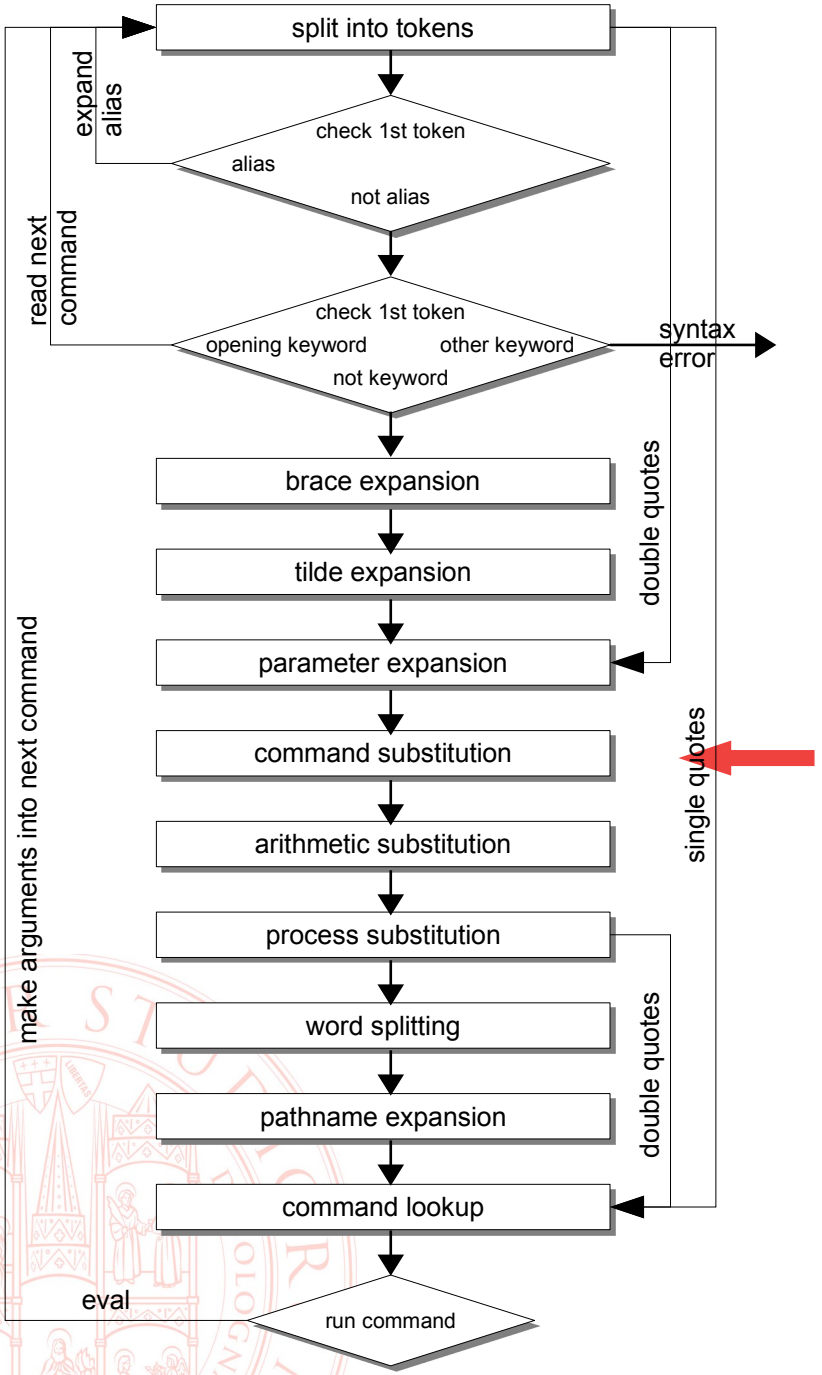
- Non eseguito sulle parti di riga racchiuse tra doppi apici



Shell expansion

6. Parameter expansion

- Il carattere "$" può marcare l'inizio di diverse espansioni
  - parameter expansion
  - command substitution
  - arithmetic expansion
- L'esempio più semplice di PE è la sostituzione della stringa `$NAME` con il valore contenuto nella variabile `NAME`
- Questi quattro passaggi (6..9) sono eseguiti anche sulle parti di riga racchiuse tra doppi apici

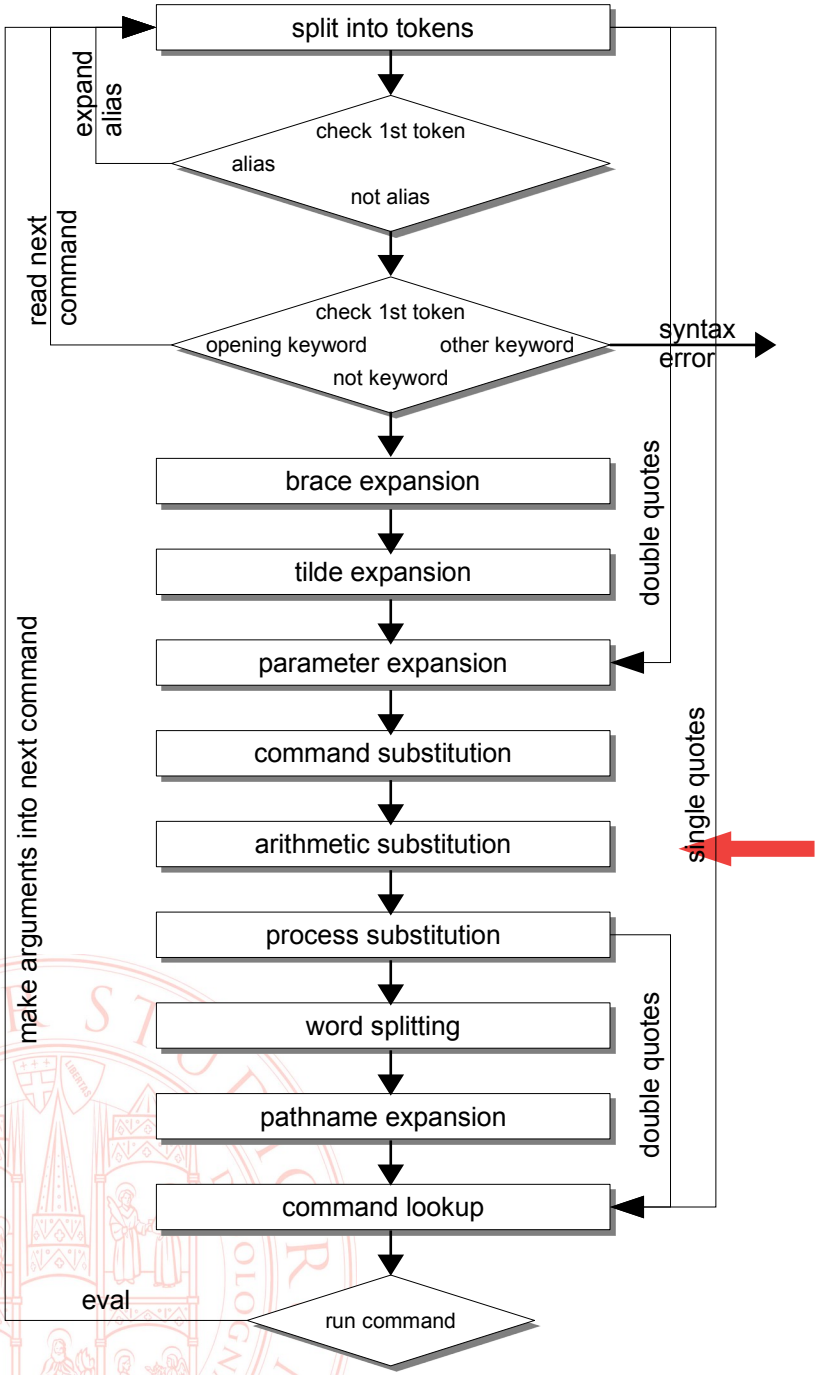


Shell expansion

7. command substitution

- il token `$(comando)` ha questo effetto:
  - viene creata una subshell
  - vi viene eseguito comando
  - sdtout di `comando` viene posto sulla riga di comando al posto del token originale, a parte eventuali righe vuote alla fine

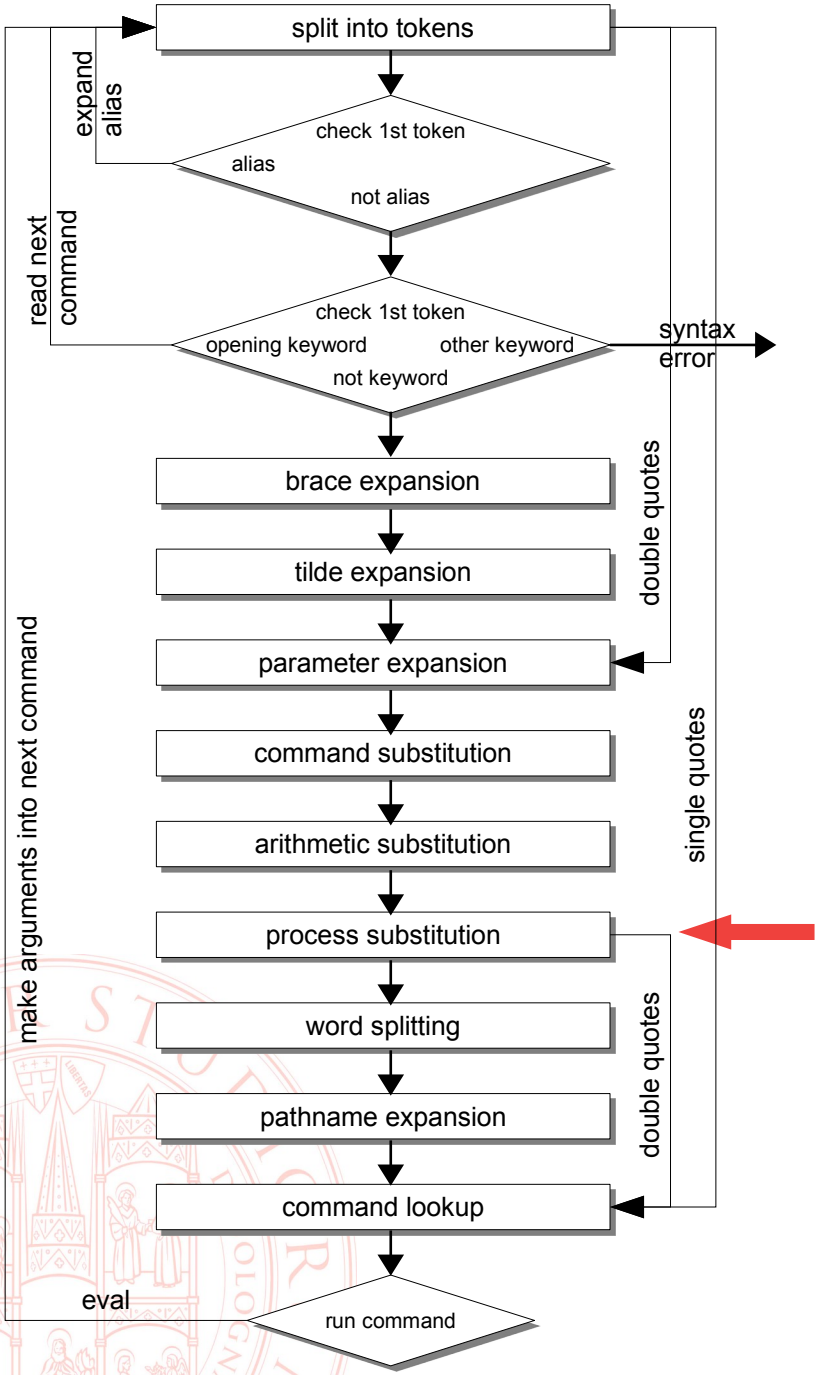
- Questi quattro passaggi (6..9) sono eseguiti anche sulle parti di riga racchiuse tra doppi apici



Shell expansion

8. arithmetic expansion

- il token `((expr))` causa la valutazione di `expr`, un'espressione aritmetica
  - se preceduto da `$`, il risultato viene posto sulla riga di comando, altrimenti l'unico effetto è eventualmente sulle variabili
- `expr` viene trattata come se fosse racchiusa tra doppi apici (quindi subisce solo i passi 6 e 7)
- Questi quattro passaggi (6..9) sono eseguiti anche sulle parti di riga racchiuse tra doppi apici

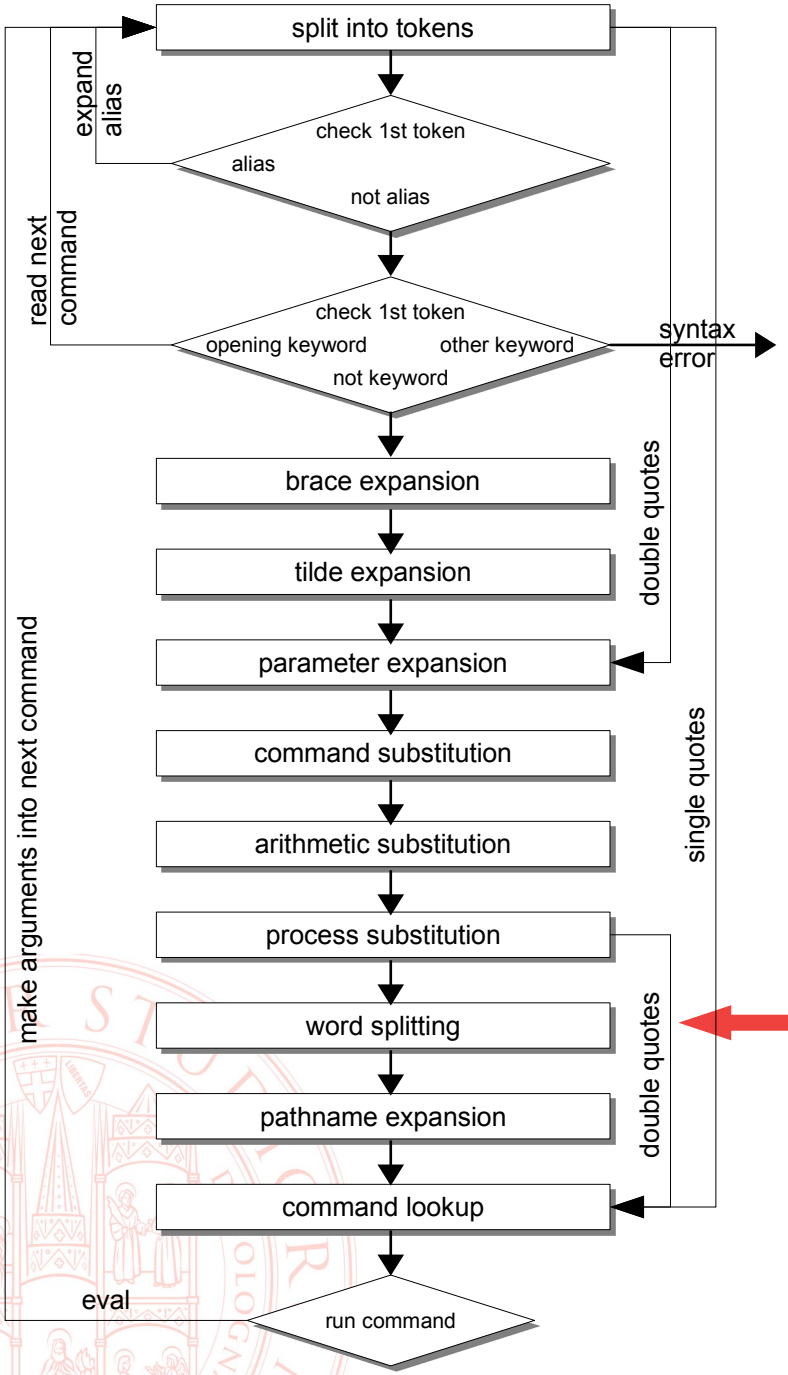


Shell expansion

9. process substitution

- il token `<(comando)` o `>(comando)` ha questo effetto:
  - viene eseguito `comando` in modo concorrente e asincrono rispetto al resto della riga
  - il suo input o output appare "come un nome di file" tra gli argomenti di tale comando

- Questi quattro passaggi (6..9) sono eseguiti anche sulle parti di riga racchiuse tra doppi apici

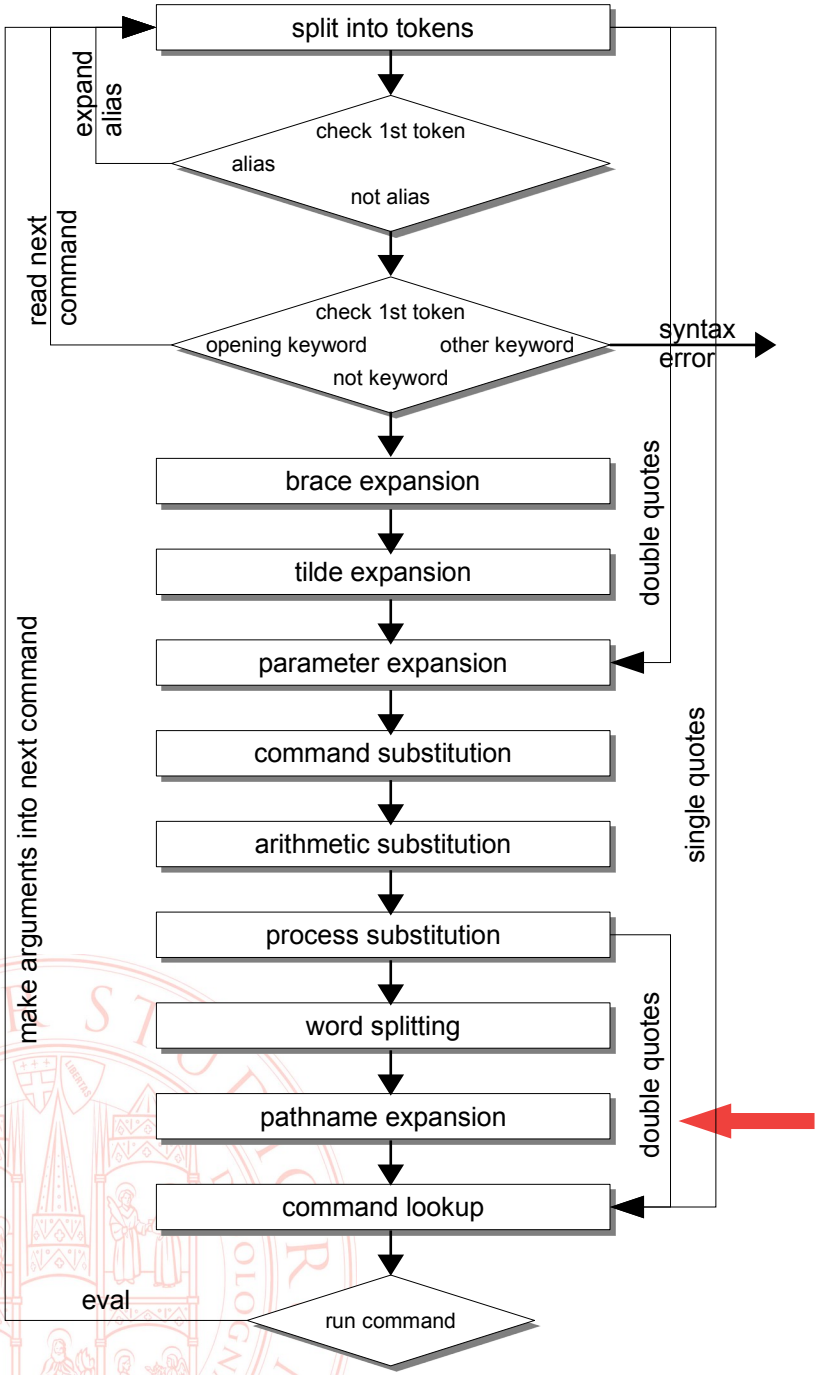


Shell expansion

10. word splitting

- I risultati dei passi 6..9 sono esaminati, e separati in *word* indipendenti
  - separatore = qualsiasi carattere presente nella variabile **IFS**
  - default IFS = `<space><tab><newline>`

- Non eseguito sulle parti di riga racchiuse tra doppi apici



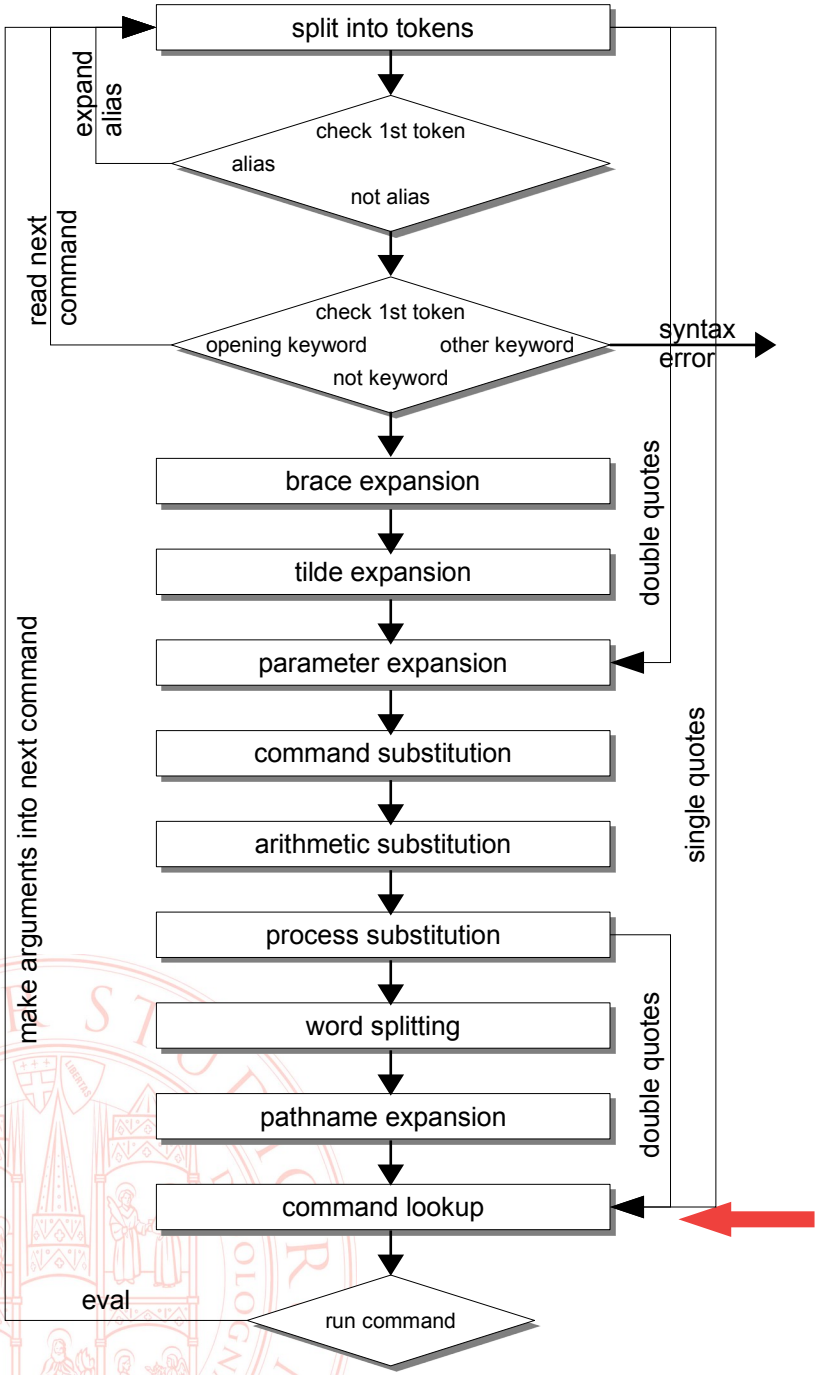
Shell expansion

11. pathname expansion

- Ogni word viene esaminata e se contiene uno dei caratteri
  - *
  - ?
  - [

  viene considerata un *pattern* e sostituita con tutti i nomi di file che concordano

- Non eseguito sulle parti di riga racchiuse tra doppi apici



Shell expansion

12. quote removal ed esecuzione

- Vengono rimosse tutte le occorrenze di caratteri di quoting “usate” effettivamente
  - non protette da altri quoting
  - non generate dai passi 6..9
- Vengono impostati gli stream in caso di ridirezione
- Viene cercato il comando in quest’ordine
  - funzioni
  - builtin
  - eseguibili in $PATH

Quoting

- In sintesi: Il meccanismo di espansione di wildcard e variabili è potente ma interferisce con l'interpretazione **letterale** di alcuni simboli: `[ ] ! * ? $ { } ( ) " ' ` \ | > < ;`

- Quando si debbano passare come parametro ad un comando delle stringhe contenenti tali simboli, è necessario **proteggerli dall'espansione**.

Quoting – metodi

\ (backslash) inibisce l'interpretazione del solo carattere successivo come speciale

' (apice) ogni carattere di una stringa racchiusa tra una coppia di apici viene protetto dall'espansione e trattato letteralmente, **senza eccezioni**.

" (apice doppio o virgolette) ogni carattere di una stringa racchiusa tra una coppia di virgolette viene protetto dall'espansione, con **l'eccezione** del **$**, del backtick (`), di ****, ed altri casi particolari

Quoting – osservazioni

- I simboli di quoting in quanto speciali essi stessi vanno protetti dall'espansione se serve utilizzarli per il loro valore letterale, ad esempio
  - `\"` il backslash protegge le virgolette → sulla riga resta `"`
  - `'"'` come sopra, gli apici proteggono le virgolette
  - `\\` il primo backslash protegge il secondo → sulla riga resta `\`
- In una riga di comando si possono mescolare frammenti protetti in modo diverso, verranno semplicemente concatenati dopo l'espansione e la quote removal
  - es. `"protetto da virgolette"\*'o da apici'`
    sarà espanso come singolo token di valore
    `protetto da virgolette*o da apici`

Comando echo

- Per visualizzare un messaggio, bash mette a diposizione il builtin echo che stampa i caratteri che lo seguono. Questo è immediatamente utilizzabile per visualizzare il valore di una variabile o il risultato di una pathname expansion
  - (memento: *sfrutta il meccanismo di espansione di bash*, **echo** non sa cosa sia una variabile, né un pathname)

- `echo PATH`
  - visualizza "PATH"
- `echo $PATH`
  - visualizza il contenuto della variabile PATH
- `echo *`
  - visualizza tutti i nomi di file nella directory corrente

Pathname expansion

- Doppiamente importante:
  - molto usato interattivamente coi comandi di gestione file e directory
  - Una delle ultime espansioni svolte da bash: opera quindi su stringhe che potrebbero essere state generate dai passi precedenti
- I pattern qui descritti vengono confrontati col filesystem
  - se non esistono file che corrispondono al pattern, questo resta inalterato sulla riga di comando
  - se esistono file che corrispondono, al posto del pattern vengono posti tutti i loro nomi, in ordine alfabetico
- Ci sono varie eccezioni legate al “punto” iniziale e a opzioni della shell – vedere la sezione *pathname expansion* di man bash(1), ma il funzionamento base è illustrato qui di seguito

Pattern per pathname expansion

- `*` rappresenta una qualunque stringa di zero o più caratteri
- `?` rappresenta un qualunque carattere singolo
- `[SET]` rappresenta un qualunque carattere appartenente a **SET**
  - SET può essere un elenco, es. `[afhOV]`
  - SET può essere un intervallo, es. `[a-k]`
    - l'ordine dipende dal *locale*
    - più intervalli si possono unire con la virgola es. `[a-d,0-5]`
  - SET può essere negato con `!` o `^`, es `[!a]` `[^A-Z]`
  - SET può essere una classe come per egrep, es. `[[:alnum:]]`
  - per includere `-` o `]` nel SET, metterli come primo carattere

Esempi di pathname expansion

- `echo *`
  - elenca i file del direttorio corrente
- `echo [a-p,1-7]*[cfd]?`
  - elenca i file i cui nomi hanno come iniziale un carattere compreso tra **a** e **p** o tra **1** e **7**, se il penultimo carattere è **c**, **f**, oppure **d**.
- `echo \*`
  - esegue l'echo del carattere *, privato del suo significato di wildcard
- `echo *[!\*\?]*`
  - elenca tutti i file del direttorio corrente che abbiano almeni un carattere diverso dalle wildcard * e ?
- `echo /*/*/*`
  - elenca tutti i file dei direttori di secondo livello a partire dalla root
- *nota*: è possibile abilitare comportamenti più complessi (Es. moltiplicatori, inversione del matching) attivando l'opzione extglob con `shopt`

Brace expansion

- È un meccanismo di espansione per generare sequenze di stringhe secondo un pattern con la stessa sintassi della pathname expansion, ma **le stringhe sono generate indipendentemente dal fatto che esistano o meno file che rispettano il pattern**
- Sintassi:

  `[PRE]{LISTA}[POST]` oppure `[PRE]{SEQUENZA}[POST]`
- Esempio con lista:

  `a{d,c,b}e` → espanso dalla shell in `ade ace abe`
- Esempi con sequenza:

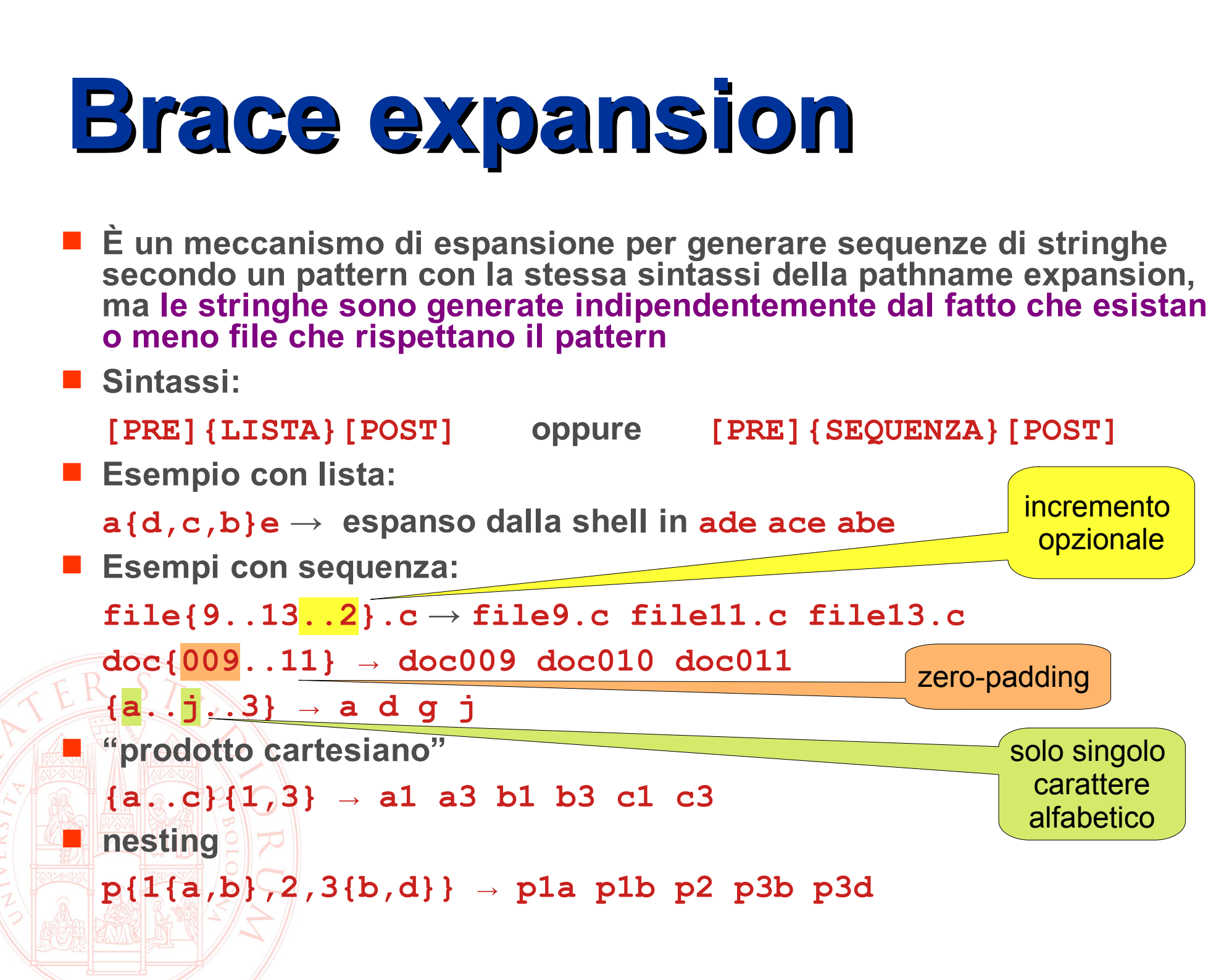
  `file{9..13..2}.c` → `file9.c file11.c file13.c`

  `doc{009..11}` → `doc009 doc010 doc011`

  `{a..j..3}` → `a d g j`
- "prodotto cartesiano"

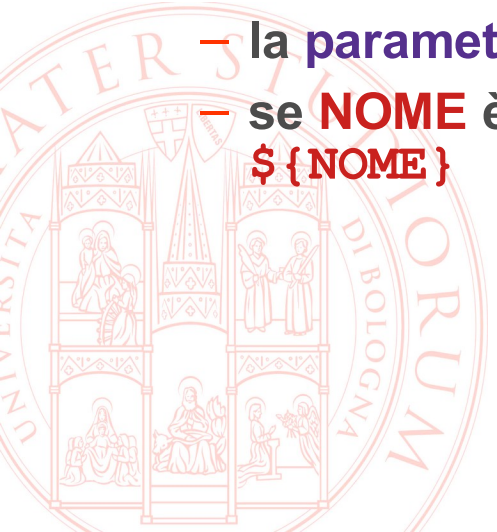
  `{a..c}{1,3}` → `a1 a3 b1 b3 c1 c3`
- nesting

  `p{1{a,b},2,3{b,d}}` → `p1a p1b p2 p3b p3d`



Variabili

- Le variabili sono un modo offerto dalla shell per memorizzare delle stringhe di testo sotto un dato nome.
- La modifica o la creazione di una variabile si ottengono semplicemente indicando sulla command line il nome della variabile seguito da = e dal valore che le si vuole attribuire.
  - Nota: non inserire spazi prima o dopo il simbolo =
  - Perché?
- Es: `pippo=valore`
  - assegna "valore" alla variabile pippo
- Utilizzo delle variabili
  - la parameter expansion rimpiazza `$NOME` col valore della variabile NOME
  - se NOME è composto o ambiguo, lo si protegga con `{}` – si utilizzi cioè `${NOME}`



Quoting – esempi di interazione

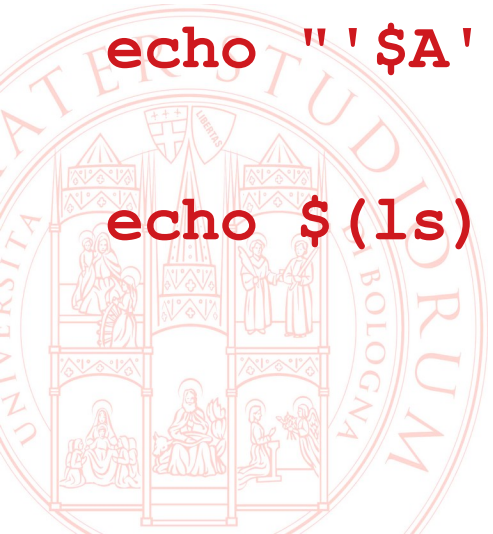
`ls *\**` lista i nomi dei file che contengono il carattere * in qualunque posizione

`echo "$A"` stampa esattamente il contenuto della variabile A (inibizione dei passi successivi alla parameter expansion)

`echo '$A'` stampa esattamente $A
(inibizione di quasi tutti i passi, inclusa parameter expansion)

`echo "'$A'"` chi vince?

`echo $(ls) ; echo "$(ls)"` dov'è la differenza?



Variabili di ambiente

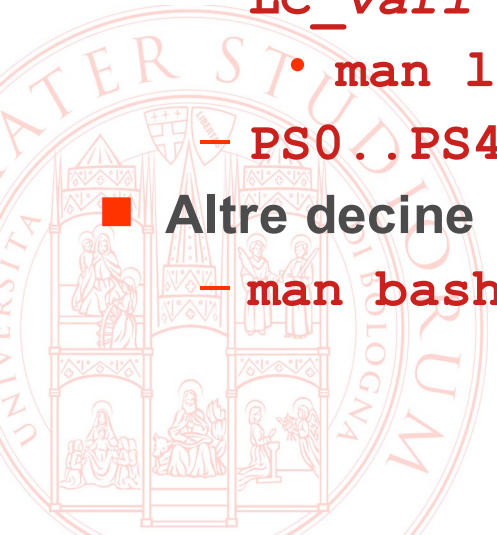
- Vi sono alcuni dati, solitamente riguardanti il sistema o le preferenze di un utente, che sono utili a tutti i comandi (es. la versione del sistema operativo, la lingua dell'utente, ecc...).
- Sarebbe faticoso e ripetitivo doverli passare come argomento ad ogni comando che si esegue: per questo unix dispone del meccanismo delle variabili di ambiente.
- La shell distingue le variabili d'ambiente dalle variabili standard (che invece rimangono confinate alla shell stessa) per mezzo dell'esportazione.

```
export pippo
```

- L'ambiente (environment) corrente può essere visualizzato col comando **set**
  - come si può notare non contiene solo variabili
- Un assegnamento prima di un comando modifica solo per tale esecuzione l'ambiente
- Il comando **env** permette un controllo più preciso

Variabili notevoli per bash

- Settate da bash:
  - `BASHPID` PID della shell corrente
  - `$` PID della shell “capostipite”
  - `PPID` PID del parent process della shell “capostipite”
  - `HOSTNAME` nome dell’host
  - `RANDOM` un numero casuale tra 0 e 32767
  - `UID` id utente che esegue la shell
- Usate da bash:
  - `HOME` home directory dell’utente
  - `LC_`*`vari`* scelta dei vari aspetti della localizzazione
    - `man locale(7)`, anche `locale(1)` e `locale(5)`
  - `PS0..PS4` prompt in diversi contesti
- Altre decine
  - `man bash` → `Shell variables`



Variabili posizionali

- Ogni script può accedere agli argomenti indicati sulla propria linea di comando, utilizzando le variabili che hanno per nome un numero
- `$0`, `$1`, `$2`, ...

  → comando, primo arg, secondo arg, …
  - utilizzare obbligatoriamente `${NUM}` se **NUM** > 9
- `$*` e `$@`

  vengono espanse in `$1 $2` ... ma questo può causare problemi se gli argomenti contengono caratteri speciali, invece:
- `"$*"` e `"$@"`

  vengono espansi in `"$1 $2 ..."` e `"$1" "$2" ...`
- `$#`

  contiene il numero di argomenti

Variabili posizionali

- Le variabili posizionali possono essere settate manualmente, per testare la correttezza di una riga di comando senza bisogno di inserirla in uno script a cui passare parametri

```
set pippo pluto paperino
echo $2
pluto
```

- Il comando **shift** riduce per scorrimento l'elenco delle variabili posizionali (esclusa `$0`), assegnando a $N il contenuto di $N+1 (il valore di $1 verrà quindi perso e $# sarà decrementato di 1)

```
echo $# $1 $2 $3
3 pippo pluto paperino
shift
echo $# $1 $2
2 pluto paperino
```

Parameter expansion
Assegnamento condizionale a variabili

- Quando si lavora con le variabili fornite sulla linea di comando, più spesso che in altri casi, è utile poter gestire facilmente l'assegnazione di valori di default
- `FILEDIR=${1:-"/tmp"}`

  usa il default **/tmp** se $1 è null o not set – non altera $1
- `cd ${HOME:=/tmp}`

  se $HOME è null o not set, viene inizializzata al default **/tmp**, poi in ogni caso espansa
- `FILESRC=${2:?"Error. You must supply a source file."}`

  stampa l'errore ed esce se $2 è null o not set
  - Nota: omettendo il ":", la stringa vuota viene considerata valida ed il valore di default è usato solo se la variabile fornita è not set

Parameter expansion
Manipolazione di variabili / search&replace

name:number:number	Substring starting character, length
#name	Return the length of the string
name#pattern	Remove (shortest) front-anchored pattern
name##pattern	Remove (longest) front-anchored pattern
name%pattern	Remove (shortest) rear-anchored pattern
name%%pattern	Remove (longest) rear-anchored pattern
name/pattern/string	Replace first occurrence
name//pattern/string	Replace all occurrences

Es. cambio l'estensione .bad di un file in .good

```
mv "${FN}" "${FN/.bad/.good}"
```

- **Molti altri usi della parameter expansion si trovano nell'omonima sezione della man page bash(1)**

Accesso indiretto

- Un'espansione particolare permette di ottenere il nome di una variabile, che poi può essere usato in qualsiasi altra espressione, ottenendo un riferimenti indiretto ai valori:

```
CHIAVE=PIPPO
PIPPO=VALORE
echo ${!CHIAVE}
VALORE
```

Array con indice numerico

- Dichiarazione (non obbligatoria)

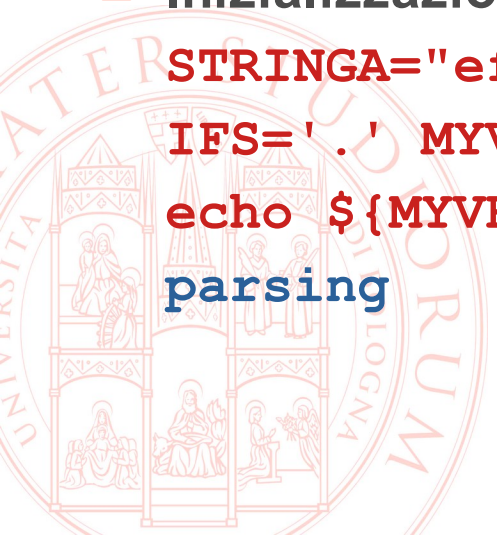
```
declare -a MYVECTOR
```

- Accesso a singole celle
  - assegnamento `A[0]="primo valore"`
  - accesso `echo ${A[0]}` ← brace: evita confusione di [] con pathname
- Inizializzazione diretta

```
MYVECTOR=(un elenco di elementi)
echo ${MYVECTOR[2]}
di
```

- Inizializzazione con separatori alternativi

```
STRINGA="effettua.il.parsing.come.read"
IFS='.' MYVECTOR=($STRINGA)
echo ${MYVECTOR[2]}
parsing
```



Array con indice numerico

- Gli indici possono essere non consecutivi!

```
A[5]="un elemento"
A[8]="un altro elemento"
```

- Per visualizzare tutti gli elementi dell'array si può usare l'indice * o @
  - La differenza di comportamento è la stessa che c'è tra $* e $@ quando virgolettati

```
echo "${A[*]}"
un elemento un altro elemento
```

- Per conoscere il set di indici corrispondenti a celle effettivamente assegnate dell'array, si utilizza ${!name[@]}

```
echo ${!A[@]}
5 8
```

- Per conoscere il numero di celle assegnate si utilizza ${#name[*]}

```
echo ${#A[*]}
2
```

Array associativi (bash 4 e successive)

- Negli array associativi l'indice può essere una stringa, non solo un numero: sono mappe chiave-valore

```
declare -A ASAR
ASAR[chiaveuno]=valoreuno
echo ${ASAR[chiaveuno]}
valoreuno
KEY=chiaveuno
echo ${ASAR[$KEY]}
valoreuno
```

Il builtin read

- **read** legge stringhe da stdin e le assegna a variabili.
  - L'input viene tokenizzato usando IFS (di default qualsiasi spaziatore)
  - Se ci sono più token che variabili, quelli in eccesso finiscono tutti nell'ultima variabile specificata, come unica stringa, separatori inclusi

```
read A B C
oggi ho portato un panino per pranzo
echo $A / $B / $C
oggi / ho / portato un panino per pranzo
```

- read separa utilizzando IFS

```
IFS=: read A B C
oggi:ho portato:un panino:per pranzo
echo $A / $B / $C
oggi / ho portato / un panino:per pranzo
```

Il builtin read

- *memento*: read è un builtin → documentato con **help read**
- alcune opzioni utili:
  - `-p PROMPT` stampa **PROMPT** prima di accettare input
  - `-u FD` legge da **FD** invece che da stdin
  - `-a ARRAY` assegna i token a elementi di **ARRAY**

```
read -p "dimmi tre colori: " -a COL
dimmi tre colori: rosso verde blu
echo ${COL[1]}
verde
```

come sempre
in blu l'output dei comandi
in verde l'input dell'utente

Il builtin read – memento processi!

- `echo ciao | read A`

  `echo $A`

  (nulla)
  - perché?
- Quando si realizzano script, tenere sempre conto di quali sottoprocessi si generano con le pipe. Problemi comuni:
  - manipolare variabili nei processi figli senza perdere i risultati prima di poterli utilizzare
    - possibile soluzione: subshell
    - `echo ciao | ( read A ; echo $A )`
  - necessità di usare read per acquisire dati interattivamente dall'utente in un processo figlio che ha stdin alimentato da una pipe anziché da terminale
    - possibile soluzione: creazione di un file descriptor per il terminale

```
exec 3<$(tty)
echo ciao | ( read -u 3 A ; echo $A )
```

il comando tty restituisce il file speciale che descrive il terminale connesso a stdin (es. /dev/pts/0)

Fare i conti con la shell

- Bash tratta tutto come stringhe salvo in alcuni contesti particolari, nei quali può svolgere operazioni aritmetiche
  - su numeri **interi**
    - per calcoli a precisione arbitraria si veda `man bc(1)`
  - senza dare errori in caso di overflow
    - dimensione: signed int dell'architettura
- Contesti in cui avviene la *arithmetic evaluation*:
  - all'assegnamento di valori su variabili **dichiarate intere**

```
declare -i N
N="3 * (2 + 5)"
echo $N
21
```

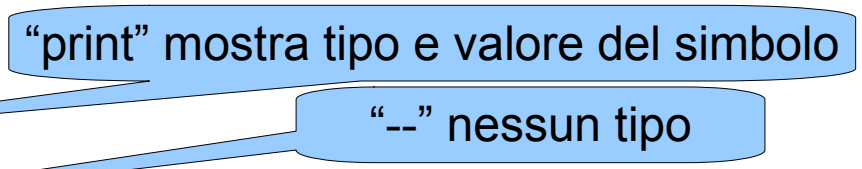
  - usando il builtin **let** o l'equivalente comando composto **(( ))**

```
let N++
echo $N
22
```

    - note: l'espressione viene valutata, produce effetti sulle variabili, **ritorna true se il risultato non è nullo**, produce errori su stderr ma nulla su stdout

Arithmetic evaluation / expansion

- (( ))
  - si comporta come " " nel proteggere gli elementi dell'espressione
  - riconosce le variabili per nome
    - anche se non sono dichiarate intere
    - senza bisogno dell'espansione bash (quindi senza prefisso $)
      - attenzione: l'espansione è ammessa ma in tal caso non vale quanto segue:
    - interpretandole a valore zero se non definite
  - **il token $(( )) viene espanso col risultato dell'espressione**
- Esempio




```
counter=0
declare -p counter
declare -- counter="0"
echo $(( counter++ )) $(( counter++ )) $(( counter++ ))
0 1 2
counter=$(( counter * newvar + 100 ))
echo $counter
100
```

Operatori e basi

- Gli operatori sono sostanzialmente quelli del C

`id++ id-- ++id --id`	post/pre-incremento/decremento
`+ - * /`	somma sottrazione prodotto divisione
`** %`	elevamento a potenza, modulo
`! ~ & | ^`	NOT NOT AND OR XOR bit-a-bit
`<< >>`	shift binario a sinistra / destra
`= *= /= %= += -= <<= >>= &= ^= |=`	assegnamento
`<= >= < > == !=`	confronto
`&& ||`	AND / OR logico
`expr1?expr2:expr 3`	restituisce il risultato di expr2 o expr3 rispettivamente se expr1 è true o false

- I numeri possono essere espressi in qualsiasi base tra 2 e 64
  - default `10`, prefisso `0` → ottale, prefisso `0x` → esadecimale
  - prefisso `B#` → base `B`
    - cifre utilizzabili: `0..9a..zA..Z@_`
    - per `B`<=36, `a..z` e `A..Z` sono intercambiabili

Controllo di flusso

- Sequenze
- Funzioni
- Verifica di condizioni
- Esecuzione di alternative
- Cicli definiti e indefiniti

Sequenze di comandi

- Come già accennato, è possibile raggruppare comandi per trattarli come un singolo processo aprendo una subshell con `( )`
- Se si vuole creare una sequenza senza aprire una subshell, per mantenere l'esecuzione nello spazio di memoria della shell principale, si usa `{ }`
  - ogni comando nella sequenza deve essere terminato da **newline** o `;`
  - Esempio:

```
{ read A ; read B ; } <<< e$'\x0a'f
echo $A . $B
e . f
```

produce ASCII newline
man bash → $'string'

- In entrambi i casi, l'exit code della sequenza è quello dell'ultimo comando eseguito

Funzioni

- Le funzioni sono sequenze di comandi con un nome

  `function NOME() { SEQUENZA ; }`
- Possono ricevere parametri
  - utilizzabili come i parametri posizionali dello script **$1**, **$2**, ...
    - che divengono inaccessibili
    - solo **$0** resta impostato al nome script
  - il nome della funzione viene posto in **$FUNCNAME**
- Se invocate "semplicemente", sono eseguite nel contesto del chiamante
  - stesso spazio di memoria
  - variabili "globali"
  - possibilità di dichiarare variabili locali con `local`
- **Attenzione, al solito, alle invocazioni in pipeline** (la funzione sarà eseguita dalla bash figlia creata automaticamente)

Valutazione di condizioni

- I comandi di controllo di flusso di bash non elaborano espressioni logiche, ma decidono il percorso sulla base dell'exit code di un processo (0==true, altri valori==false)
  - nota: l'exit code di un processo si trova nella variabile speciale $? settata dalla shell al termine del processo
- Molto comodo, ma a volte serve un interprete di espressioni logiche: per questo esistono diversi builtin e comandi
  - Builtin: `test` , `[ ]` , `[[ ]]`
  - Comandi `test` , `[ ]`
- Come sempre, i builtin vengono eseguiti prima di cercare comandi equivalenti.
  - Faremo quindi riferimento ai builtin
  - I comandi esterni sono standard su molti sistemi indipendentemente dalla shell usata. Tipicamente hanno sintassi identiche. Utile ricordarlo se serve portabilità.

test / [ ]

- **test** e **[** sono lo stesso builtin
  - seguiti da un'espressione la valutano e ritornano 0 o 1 coerentemente al risultato (true o false)
  - l'unica differenza è che **[** esige come ultimo parametro **]**
    - pura questione di leggibilità

- Principali test unari

  `test OP ARG`

  - su stringhe

  `-z` true se la stringa è vuota

  `-n` true se string è non vuota

  - su file

  `-e` true se file esiste

  `-f` true se è un file regolare

  `-d` true se è una directory

  `-s` true se file non è vuoto

  (altri 20 su help test)

- Principali test binari

  `test ARG1 OP ARG2`

  - confronto **lessicale** tra stringhe

  `=` , `!=` , `<` , `>`

  - confronto **numerico** tra stringhe

  `-eq` , `-ne` equal, not equal

  `-lt` , `-le` less than , less or equal

  `-gt` , `-ge` greater than, greater or equal

  - confronto tra file

  `-nt` newer than

  `-ot` older than

[[ ]]

- Nelle versioni più recenti di bash è disponibile questo builtin
- Supporta le stesse funzioni di **test** / **[** e inoltre
  - gli operatori binari `==` e `!=` fanno match del parametro di sinistra con un pattern espresso dal parametro di destra con la stessa sintassi della **pathname expansion** (**[[ ]]** equivale a un quoting, quindi i metacaratteri non vanno protetti dall'espansione shell)

    Es. `[[ "ciao" == c?a[l-z] ]]` → true
  - l'operatore binario `=~` fa match del parametro di sinistra con una **regular expression** specificata dal parametro di destra

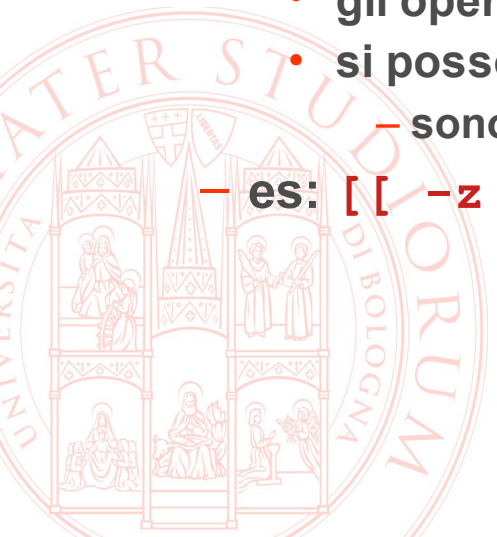
    Es. `[[ "ciao" =~ ^c.{2}o$ ]]` → true

Combinazione di test logici

- è possibile combinare controlli elementari in espressioni più complesse utilizzando i classici operatori logici

  - con `test` / `[ ]`
    - gli operatori **AND**, **OR**, **NOT** sono rispettivamente `-a` , `-o` , `!`
    - si possono effettuare raggruppamenti con le parentesi tonde
      - vanno protette dall'espansione della shell → usare `\( \)`
  - es: `test -z "$1" -a \( f1 -nt f2 -o "$A" -eq "1" \)`

  - con `[[ ]]`
    - gli operatori **AND**, **OR**, **NOT** sono rispettivamente `&&` , `||` , `!`
    - si possono effettuare raggruppamenti con le parentesi tonde
      - sono automaticamente protette da `[[ ]]` → usare `( )`
  - es: `[[ -z "$1" && ( f1 -nt f2 || "$A" -eq "1" ) ]]`



Combinazione di test logici tra processi

- gli operatori `&&`, `||`, `!` si possono usare anche sulla riga di comando per combinare logicamente gli exit code di qualsiasi processo
  - Es: `cd "$MYDIR" && ls "$MYFILE"`
    → **true** se riesco a entrare in **$MYDIR** e riesco a elencare **$MYFILE**
- Sia in questo contesto, che nel caso di uso all'interno di **[[ ]]**, sono **valutati in modo efficiente**: se il primo operando è sufficiente a determinare il risultato, il secondo non viene valutato - nel caso dei processi questo equivale a un "if"
  - nell'esempio precedente, se **cd** fallisce, **ls** non viene nemmeno lanciato
  - altro esempio: `cd "$MYDIR" || echo "$MYDIR inaccessibile"`
    - solo se **cd** fallisce è necessario eseguire il secondo comando per determinare se il risultato dell'OR è true o false

if

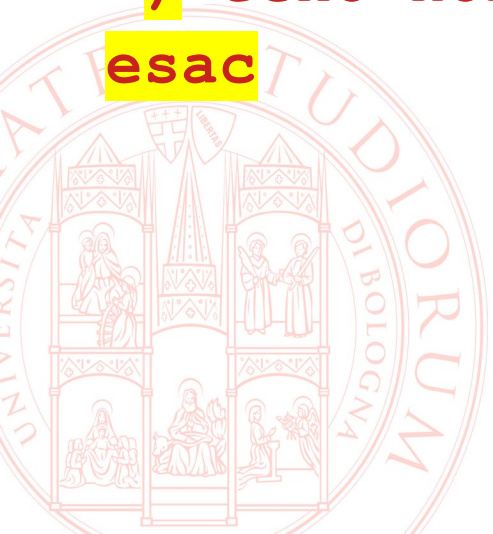
```
if COMANDO1
then
     comandi eseguiti se COMANDO1 ritorna true
[ elif COMANDO2
then
     comandi eseguiti se COMANDO2 ritorna true ]
[ else
     comandi eseguiti se nessun ritorno true ]
fi
```

- naturalmente i **COMANDI** possono essere composti (sequenze, subshell, combinazioni logiche, ecc.)

case

- La verifica di condizioni multiple è più leggibile, rispetto a una catena di **elif**, con il costrutto **case**:

```
case "$variabile" in
nome1) echo vale nome1 ;;
nome?) echo vale nome2, nomea, nomez ;;
nome*) echo vale nome11, nome, nomepippo ;;
[1-9]nome) echo vale 1nome, 2nome, …, 9nome ;;
*) echo non cade in nessuna delle precedenti ;;
esac
```



Cicli definiti

- La keyword for itera su di una lista di elementi

  `for NAME [in WORDS ... ] ; do COMMANDS; done`

- Tipici casi d'uso:

  WORDS = pattern di pathname expansion
  - itera direttamente sui nomi di file prodotti dall'espansione
  - es. `for F in /tmp/*.bak ; do rm -f "$F" ; done`

  WORDS = parametri della command line
  - es. `for PAR in "$@" ; do echo "$PAR" ; done`
  - (o di qualsiasi array con `"${ARR[@]}"`)

  WORDS = *command substitution*
  - es. `for USER in $(cat /etc/passwd | cut -f1 -d:)` ...

  WORDS = *brace expansion*
  - es. `for ITEM in item_{a..z}` ...

Cicli definiti

- Modi più flessibili di generare sequenze numeriche:

```
for BACKWARDSTENTHS in $(seq  1 -0.1  0) …
```

start | incremento | end

- Sintassi C-like nelle versioni recenti di bash:

```
for (( i=0, j=0 ; i+j < 10 ; i++, j+=2 ))
```

Espressioni di inizializzazione

Test di terminazione

Il ciclo è reiterato se

- c'è un'espressione logica che risulta true
- c'è un'espressione aritmetica che dà risultato zero

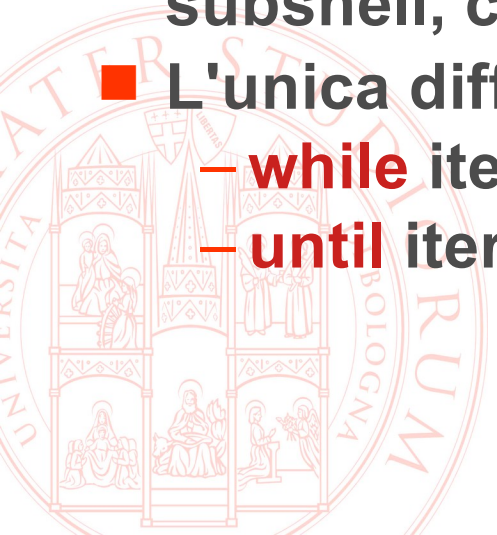
attenzione, se è presente più di un test, sono eseguiti tutti ma solo l'ultimo determina se il ciclo prosegue

Espressioni eseguite ad ogni iterazione

Cicli indefiniti

```
while COMANDO     oppure      until COMANDO
do
    LISTA
    COMANDI
    ITERATI
done
```

- naturalmente **COMANDO** può essere composto (sequenze, subshell, combinazioni logiche, ecc.)
- L'unica differenza tra i due è
  - **while** itera se **COMANDO** restituisce true
  - **until** itera se **COMANDO** restituisce false



Terminazione anticipata di cicli o iterazioni

- `break [N]`
  - esce da un ciclo for, while o until
  - se specificato **N**, esce da **N** cicli annidati
- `continue [N]`
  - salta alla successiva (possibile) iterazione di un ciclo for, while o until
  - se specificato **N**, riparte risalendo di **N** cicli annidati

Accorgimenti utili

- condividere definizioni
- Interpretare una stringa come una riga di comando
- semplificare il parsing di opzioni
- operare con identità e privilegi differenziati
- gestire il tempo
- creare file temporanei

Condividere variabili e altre definizioni

- il comando `source` può essere usato per eseguire uno script nel contesto di un altro (inclusa la riga di comando interattiva)
- es.:
  - supponiamo che lo script `common.sh` contenga assegnamenti di valori a variabili, definizioni di funzioni e di alias
  - dopo l'esecuzione di `source common.sh` le variabili, funzioni e alias saranno definite anche nello script "chiamante"
- utile per condividere parametri tra script correlati tra loro e per creare librerie di funzioni importabili

Interpretare una stringa come una riga di comando

- Il builtin **eval** permette di processare un file come se fosse uno script, sottoponendolo ai 12 passi di valutazione elencati in precedenza.
- Questo permette ad uno script di generare altri script ed eseguirli correttamente.

Es:

```
listpage="ls | more"
$listpage
ls: cannot access |: No such file or directory
ls: cannot access more: No such file or directory
eval $listpage
… elenco file paginato …
```

La parameter expansion avviene dopo la tokenization... "|" e "more" appaiono quindi troppo tardi per essere interpretati come token generici e non solo come argomenti di ls

Interpretare una stringa come una riga di comando

- Una delle funzionalità più utili, dopo la ri-valutazione dei separatori, è far comparire variabili e forzarne l'espansione.
- Bisogna, come sempre, porre molta attenzione all'escaping dei metacaratteri per inibire la loro interpretazione al primo passaggio di espansione ed eventualmente abilitarli al secondo.
- Esempio:

```
A=ciao
eval "P=$A ; echo $P"
(nulla)
unset P
eval "P=$A ; echo \$P"
ciao
```

$P è espanso dalla shell chiamante, in cui non ha alcun valore

\$P è espanso dalla shell chiamante in **$P**

```
eval P=ciao ; echo $P
```

select

- È un builtin utile per semplificare la selezione tra alternative

```
directorylist="Finished $(for i in /*;do [ -d "$i" ] && echo $i; done)"
PS3='Directory to process? ' # Set a useful select prompt
until [ "$directory" == "Finished" ]; do
    printf "%b" "\a\n\nSelect a directory to process:\n" >&2
    select directory in $directorylist; do
        # User types a number which is stored in $REPLY, but select
        # returns the value of the entry
        if [ "$directory" = "Finished" ]; then
            echo "Finished processing directories."
            break
        elif [ -n "$directory" ]; then
            echo "You chose number $REPLY, processing $directory..."
            # Do something here
            break
        else
            echo "Invalid selection!"
        fi # end of handle user's selection
    done # end of select a directory
done # end of while not finished
```

getopts

- Per costruire script che supportino la classica sintassi
  `comando OPZIONI_PRECEDUTE_DAL_TRATTINO ARGOMENTI`
  è utile il builtin getopts.
- Sintassi:
  `getopts OPTSTRING NAME [arg]`

Stringa che definisce i caratteri da riconoscere come opzioni. Se un carattere è seguito da : significa che è atteso un parametro per quell'opzione

Nome di variabile in cui collocare il parametro correntemente analizzato

- Funzionamento:
  - ad ogni invocazione, getopts esamina una variabile posizionale
  - assegna il suo indice ad OPTIND ed il suo contenuto a NAME.
  - se è un'opzione che richiede un argomento (atteso nella successiva variabile posizionale) questo viene letto (incrementando OPTIND) ed assegnato a OPTARG.

getopts – una "ricetta"

```
#!/usr/bin/env bash
aflag=
bflag=
while getopts 'ab:' OPTION ; do
     case $OPTION in
         a) aflag=1
            ;;
         b) bflag=1
            bval="$OPTARG"
            ;;
         ?) printf "Usage: %s: [-a] [-b value] args\n" $(basename $0) >&2
            exit 2
            ;;
     esac
done
shift $(($OPTIND - 1)) # getopts cycles over $*, doesn't shift it

if [ "$aflag" ] ; then
     printf "Option -a specified\n"
fi
if [ "$bflag" ] ; then
      printf 'Option -b "%s" specified\n' "$bval"
fi
printf "Remaining arguments are: %s\n" "$*"
```

Esecuzione di comandi con diverse identità

- **su** (da root – da altri utenti chiederebbe la password, non pratico da scriptare)
  - **su** non è **exec**, non si mettono comandi a seguire, apre una shell e finché non si esce da quella non prosegue nel contesto chiamante
  - però accetta un comando da eseguire:

    ```
    su -c "COMANDO" - UTENTE
    ```

- **sudo COMANDO**
  - è sicuramente l'opzione da usare per eseguire comandi come root
  - richiede la configurazione di sudoers (possibilmente con **NOPASSWD**)

Misura del tempo

- **time** è una semplice keyword che anteposta a un comando ne riporta la durata

```
time ls -lR /etc/ >/dev/null 2>&1
real   0m0,155s
user   0m0,013s
sys    0m0,047s
```

Misura del tempo

- **date** è un comando ricchissimo di opzioni
  - può essere usato con l'opzione -s per impostare l'orologio del sistema
  - può essere usato per interrogare l'orologio e per convertire tra formati i *timestamp*
- Formati di output
  - **date +FORMAT** permette di selezionare cosa e come visualizzare
  - **FORMAT** è una stringa in cui vengono interpretate sequenze speciali contrassegnate dal carattere **%** ; qualche esempio dei moltissimi disponibili (→ `man date(1)`)

    `date +"%Y%m%d %H:%M:%S"` AnnoMeseGiorno Ora:Minuto:Secondo
    - `20210309 15:27:45`

    `date + "%A %e %B"` Giornosettimana Giorno Mese (locale)
    - `martedì  9 marzo`
  - uno dei formati più utili per fare calcoli è **%s**: il numero di secondi dall'epoca di Unix (1/1/70)
  - volendo più precisione, **%s%n** (appende i nanosecondi)

Misura del tempo

- Usando l'opzione -d si può fornire a date un timestamp da usare al posto del tempo corrente
- In questo modo si possono convertire i timestamp tra diversi formati, ad esempio:

```
N=$(date -d '2020-05-15 10:01' +%s)
```

  - ora di un evento interessante convertita in epoch

```
(( N+=1800 ))
```

  - aumentata di 1800 secondi

```
date -d "@$N"
```

  - stampata in modo leggibile

```
ven 15 mag 2020, 10:31:00, CEST
```

File temporanei

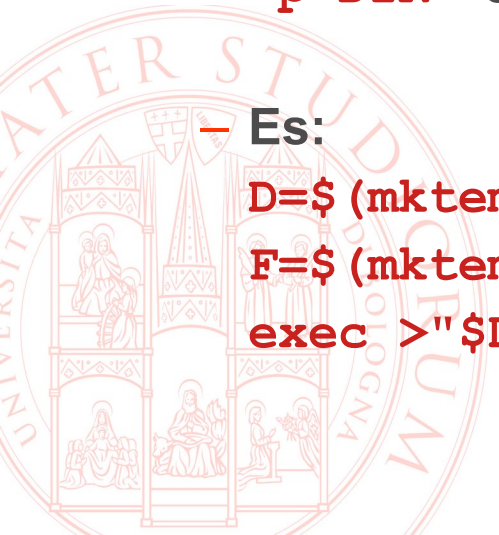
- Requisiti del nome:
  - univocità
  - non prevedibilità
    - per evitare che un intruso possa usarli per puntare a risorse sensibili
- `mktemp`
  - restituisce il nome del file creato
  - di default nel formato `/tmp/tmp.XXXXXXXXXX`
- opzioni principali

  `-d` crea una directory

  `-p DIR` crea all'interno di **DIR** invece che in **/tmp**

  - Es:

```
D=$(mktemp -d)
F=$(mktemp)
exec >"$D/$F"
```



Manipolazione nomi di file

- **basename (1)**
  - rimuove il path ed eventualmente un suffisso da un nome di file

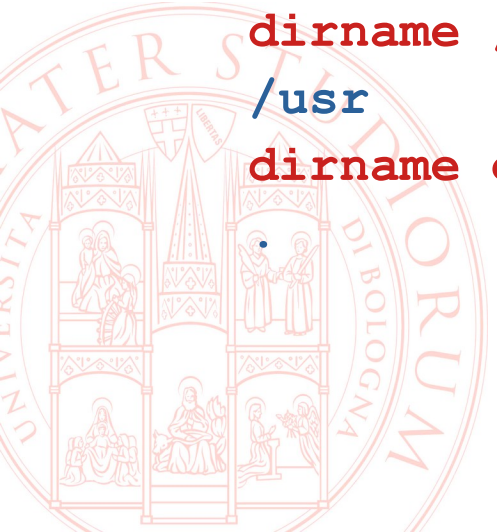
    Es.

```
basename -s .h include/stdio.h
stdio
```

- **dirname (1)**
  - rimuove l'ultimo componente del percorso da un nome di file
    - se il nome non contiene "/", restituisce "." (la directory corrente)

    Es.

```
dirname /usr/bin
/usr
dirname data.txt
.
```



printf

- builtin che permette di formattare gli argomenti in modo più complesso rispetto a echo
  - tipicamente esiste anche il comando esterno omonimo e pressochè equivalente
- Sintassi:

```
printf [-v var] format [arguments]
```

  - `-v var` assegna il risultato della formattazione alla variabile invece che produrlo su STDOUT
  - `format` è una stringa di formato documentata da **printf (1)**
    - più estensioni specifiche:

      `%b` attiva l'interpretazione delle sequenze speciali "\" negli argomenti

      `%q` aggiunge quoting in modo che il risultato sia utilizzabile su cmdline

      `%(fmt)T` produce una stringa data-ora secondo fmt (vedi strftime (3))
  - `arguments` sono gli argomenti a cui applicare la stringa di formato

script / scriptreplay

- **script (1)** crea una copia di tutto ciò che appare sul terminale
  - comandi
  - loro output
  - opzionalmente marcatori temporali della comparsa di ogni riga
    - opzione -t
- produce un file **typescript**
- termina con `Ctrl-D` o `exit`
- Il file prodotto può essere utilizzato come documentazione oppure dato come argomento a **scriptreplay (1)**
  - visualizza il typescript senza rieseguire i comandi
  - funziona solo se è stata tracciata la temporizzazione

watch

- **watch** esegue un comando periodicamente mostrando l'output sul terminale
  - non è da considerare uno strumento di automazione, ma di monitoraggio "umano" interattivo
- opzioni principali
  - `-d` evidenzia le differenze apparse dall'ultimo aggiornamento
    - `--differences=permanent` tutto ciò che è cambiato dall'avvio
  - `-n <intervallo>` imposta l'attesa tra un'esecuzione e la successiva
    - minimo 0.1 secondi
    - default 2 secondi
  - `-p` interpreta il valore passato con `-n` come periodo di aggiornamento
    - cerca di compensare il tempo di esecuzione del comando